A novel distributed architecture for IoT image processing using low-cost devices and open internet standards

Carlos A. Pérez[1], Mario S. Cleva[1], Diego O. Liska[1]

[1] CINAPTIC, Applied Research Centre of Information Technologies, Universidad Tecnológica Nacional, Facultad Regional Resistencia. French 404, (3500) Resistencia, Chaco, Argentina.

cperez@rec.utn.edu.ar,clevamario@gmail.com,
diegoorlandoliska@gmail.com

Abstract. Industry 4.0 can be defined as the integration of computers and automation to current industrial processes, with addition of smart and autonomous systems leveraged by machine learning techniques. In this scenario, a compact, dependable and fast controller is desired, featuring low energy consumption, easily programming and maintenance, with no mobile parts. Nowadays, computing power in single board computers, e.g. the Raspberry Pi among others, has been increased at a very important rate. In just three generations, Pi computers offer almost a two-fold speed gain, when compared to first models. Its design, an underlying video driver with general capabilities of regular OSes, makes them quite suitable to build image processing systems at very low cost, with no mobile parts and low energy consumption. However, designing such a system for industrial image processing is a tough challenge, since it implies to integrate cameras, image processing libraries, database servers and application software with graphical user interface, in an already resource-constrained device. This work presents a new architecture for this kind of systems, by means of open internet standards, using a self-contained, high performance web server to publish a RESTful API and a set of web pages that use latest HTML5 capabilities to manage USB webcams and system data. This proposal also integrates OpenCV as a compiled script on client-side using the new WASM paradigm, with an optimized storage for images using -industry-standard RDBMS and a modular design that can target Windows and Linux as well.

Keywords: devices, computer vision, web assembly, OpenCV, embedded

1 Introduction

Image processing has paramount importance in Industry 4.0 and IoT, which are closely related, since both share similar goals. IoT goal is to enable real world objects with speech, vision, hearing, smell and touch, so inanimate things can perform jobs more accurately and responsively, collaboratively and with automated learning [1]. Industry 4.0 is the use of those abilities in almost every stage of modern manufacturing. In this scenario, **perception** is an important layer of every IoT architecture. Computer vision is gaining widespread adoption due to proliferation of low-cost embedded

adfa, p. 1, 2011.

cameras, open source libraries for computer vision, and very affordable computing devices with diverse form factors, from smartphones to embedded systems.

Technology-wise, there are important aspects that must be taken in account when designing such a system: computing speed and capacity, human-machine interfacing, storage system, data acquisition and transmission systems, and remote access for management purposes. Most computer vision applications run as smart clients, that is, the software comprises executables based either on native binary code or using a virtual machine approach. A distributed system on a LAN can be devised, provided network bandwidth and delay are between acceptable bounds. Some solutions are now available as services on the cloud, provided by mainstream vendors such as Google, IBM and Microsoft. In these cases, however, a dependable internet connection with acceptable upload speed is mandatory, which prevents to use them in many locations in our country.

In this paper, a novel architecture for a system to computer vision is presented, built on open source hardware and software by means of a low cost, high-performance single board computer. The software comprises a highly capable, in-process, compact middleware with full HTTP stack support. This hosts a RESTful API and a set of web pages to manage image acquisition, transmission and processing. Image processing tasks, and machine learning algorithms are executed in client browser space by means of web assembly technology that provides near-native execution speed. As back-end, an open-source relational database management system (RDBMS) is used for image storage, providing single-point backup and replication. The whole system is compatible with Windows and Linux platforms, and all components can be deployed to IoT boards such as Raspberry Pi without reprogramming.

This paper is structured as follows. Section 2 contains the mainstream IoT and computer vision building blocks. In Section 3 the proposed architecture is described. Obtained results are discussed in Section 4. Section 5 presents some conclusions and possible system expansions.

2 IoT and Computer Vision building blocks

2.1 IoT layered architecture

There are four layers in IoT: perception, transmission, computing and application. **Table 1** shows the four IoT layers, with types and subtypes.

In perception layer, the most complex device is the camera, comprising several lenses and IR filters, auto focus subsystems, optical stabilization motors, and light-sensitive sensor with thousands, even millions, of picture elements. Camera uses high bandwidth, and image and video storage requirements are often the main ones, if not the only ones, that are considered for system specifications. Advanced cameras use built-in memory (up to 512 GiB) to accelerate access acquisition as much as possible.

Table 2 shows current standards for IoT components. Industry has heavily standardized internet-centric components, such as connections and protocols, that were

created in a gradual process. However, this process gained momentum in last few years, especially in HTTP open standards.

Table 1. Internet of things (IoT) layers

Layer	Features and components
Perception layer	circuits, sensors (temperature, pressure, moisture, magnetic, gyro), actuators, controllers, RFID, imaging (cameras, scanners).
Transmission layer	Connections (wired and wireless) and protocols.
Computational layer.	Hardware (Arduino, Pi, Intel Galileo, Nvidia Jetson), software (mainly IDEs such as Eclipse, Visual Studio Code, ADK), algorithms, cloud platforms (for instance, Hadoop), encryption (error correction, data security, etc.)
Application layer	For consumer and for business. Consumer includes home, lifestyle, health care, transport. Business include manufacturing, retail, public services, energy, transportation, agriculture, military, hospitality, cities.

Table 2. IoT connections and protocols

Layer	Type	Standard
Connections	Wired	USB, RJ45, ISND, IEEE 1394 (firewire).
	Wireless	Bluetooth. IEEE 802.11 Wi-Fi. IrDA. NFC. RFid. GSM. ZigBee
Protocols	Link	ARP. NDP. OSPF. Tunnels. PPP. MAC
	Internet	IPV4. IPV6. ICMP. ECN. IPsec. IGMP
	Transport	TCP. UDP. DCCP. SCTP. SCP. RSVP.
	Application	BGP. DHCP. DNS. FTP. HTTP. HTTPS. IMAP. LDAP. MGCP. MQTT. NNTP. NTP. POP.

In this work, cameras use USB 2.0 connections, wireless communication is Wi-Fi 802.11 b-g-n, internet protocols are IPV4 with TCP/IP and SCP, and applications use HTTP, HTTPS and DHCP protocols. Pages are built using latest HTML5 standards with audio/video tags for camera support.

2.2 Image processing software. Open CV.

Regarding image processing libraries, de-facto standard is OpenCV [2], latest version is 4.0 and have been ported to almost every mainstream computing platform, as well as embedded and vertical systems, ranging from native code in bare-metal machines to commercial smartphones. Basic structure in OpenCV is numerical matrix, where pixel data are stored. Since current digital photography technology can easily

take a 50-megapixel sample in lossless formats (such as DNG, digital negative), matrix memory allocation, de-allocation and image duplications are highly optimized since version 2.2. OpenCV is a modular library design, each module groups operations by type, as shown in **Table** 3. Almost all image functions in OpenCV involve matrix algebra. A morphological operation, for instance, is value calculation of a given pixel, based on values of pixels in its neighborhood. By proper selection of neighborhood shape and size, several effects can be achieved in the resulting image. This implies a heavy computation task, since the algorithm must scan all the surface of the input image, stepping pixel by pixel. This kind of complexity is optimal to be parallelized, provided the computer has more than one computing core. Because of this, OpenCV can use parallel computing in many operations.

Table 3. OpenCV modules

OpenCV modules	Features
core	Basic building blocks of the library.
imgproc	Contains manipulation functions inside OpenCV such as basic filtering, geometric image transformations, histograms, structural analysis and shape descriptors, motion analysis and object tracking, feature and object detection.
highgui	High level graphical user interface, creates and manipulates images, windows, interacts with pointer devices and keyboards, reads and writes images and videos, compatibility with Qt interfaces.
calib3d	Camera calibration and 3D reconstruction. Camera parameters inference from samples. Image rotation and shifting. Projection matrix composer and decomposer. Stereo image manipulation and camera calibration. Management of distortion in Fisheye lens.
features2d	Points detectors, descriptors, matching framework, homography (reprojection of a plane from one camera into a different camera view, by camera translation and rotation).
video	Video capture, decode and encode, video writing. Motion extraction, feature tracking, foreground extraction.
objdetect	Cascade classifier, allowing object detection by means of training with positive and negative samples.
ml	Machine learning algorithms. Normal Bayes, K-nearest, support vector machines SVM, decision trees, boosting, random tree, expectation maximization, neural networks, logistic regression.
gpu	CUDA-runtime compatible, supports Nvidia® GPUs only. Supports multiple GPUs.

OpenCV allows the use of GPUs (graphical processing units) in order to accelerate some calculations. Modern **CPUs** must retrieve input data from memory, calculate as

fast as possible, and write results back to memory. Performance is highly optimized with complex built-in circuits that support pipelines, superscalar techniques, multilevel cache, speculative execution, branch prediction, and high precision floating point arithmetic. But memory bandwidth is -comparatively- somewhat limited, and this impacts negatively the processing of large matrices and complex vector calculations. On the other hand, **GPUs** feature many computing cores, (up to thousands of them in high-end models) with limited computing power, but very high transfer rate. A *pixel shader* is a program that manipulates a pixel, performs operations to create shadows, e.g., light levels, darkness and color changes. A *vertex shader* is a conceptual extension of the former, that manipulates a set of pixels, based on a small data input. For instance, a triangle can be drawn and painted with a given color, by just defining three points in space, plus one RGB color. A vertex shader can draw this triangle, of any size, based on few data: three (`x,y`) `int32` coordinates, and three `int8` integers to describe a RGB color. By adding a third dimension for a 3D space, the pixel becomes a *vertex*, simplest entity to be managed by a 3D image algorithm. Current GPU shaders operate with 3D vertices, a vertex shader uses floating point mathematics to perform 3D to 2D projections. By decrementing the number of bits in involved `float` type, the vertex engine becomes simpler, cheaper, and resultant imagery cannot be distinguished from another made by double precision calculations.

Graphics libraries are abstractions of these GPU low-level functions [3]. By exposing them as a graphics API, they are accessible from regular programs and processes. Applications can pass data from CPU to GPU, where graphic card creates video frames and sends them to display(s). However, data in GPU memory cannot be transferred back to CPU memory. By adding a reverse pathway, GPUs can return processed data to programs and applications, thus creating the new paradigm **GPGPU** (general processing graphics card unit). Two major GPGPU APIs are OpenCL (open source) and Nvidia´s CUDA (proprietary), they both allow graphics cards to co-operate with CPUs in processing any load that comprises highly parallelizable algorithms.

2.3 Single board computer and the Raspberry PI SBC

A single board computer, or SBC, is a whole computer in just one physical circuit, featuring microprocessor, random access memory, mass memory, I/O subsystems, etc. Current models' features are small size and energy consumption, standard I/O connectors and network interfaces, and video capabilities. Since the goal of this work is a low-cost, high performance system, it is desirable to use an open source hardware and software SBC. The **Raspberry Pi** (registered trademark of Raspberry Pi Foundation) is a credit-card sized SBC that offers compatibility with 32-bit Linux (Raspbian) and Windows® 10 IoT editions, it has an important catalog of applications, utilities, programming tools and compilers, and features an ample array of industry standard connections. Due to this, the Raspberry Pi (for now of, RPI) was selected for this work.

The RPI is, conceptually, a video facility with a regular OS wrapper, quite suitable for image processing tasks. The Raspbian operating system is almost a full-fledged Linux distribution, derived from Debian, with a significative number of open-source applications, development tools, and libraries ranging from signal processing to office

suites. Latest OS images include programming IDEs, Python and C/C++ line compilers, media players, internet browsers and networking tools.

Since RPI processor has a 4-core, 4-thread design, parallel processing is possible, provided software platform supports this feature. For instance, compiling OpenCV 4.0 on the Pi Model 3B+, takes about one hour with all cores working simultaneously, resulting in almost a three-fold gain when compared to obtained figures using a single core for the same task. To prevent excessive thermal throttling, or even system crash due to heat, an active cooling system is often mounted on a resistant polymer case that protects the whole system, allowing the PI to support long lasting duty cycles.

3 A proposal for a novel system architecture

3.1 Design constraints

The aims of this proposal are as follows. Low cost is highly desirable to employ a device with open source hardware and software, to avoid license restrictions and royalties. It is also desirable a low energy consumption, in order to use a regular phone charger as power supply. In addition, system must be always performant with image manipulation. Regarding compatibility, system must accept regular peripherals (such as USB Webcams, keyboards, mice), and TCP/IP networking (wired and wireless). System must also work with mainstream libraries for image processing, namely OpenCV. Software-wise, system must be built and maintained with one development system. Operating system must run mainstream browser(s) that support HTML5 features. Tools for remote deployment, remote desktop, and secure file transfer must also be supported. HTTP server must be as compact and fast as possible, providing acceptable performance in memory-constrained devices. The RDBMS must also be tailored to fit in such constraints and must admit replication.

3.2 Selection of building blocks

In order to fulfill requirements, the following components were evaluated, and eventually selected:

3.2.1. Hardware.

Raspberry Pi Model. The **Raspberry Pi model 3B+** is the third iteration, features a Broadcom BCM2837B0 SOC (system on a chip), 1GiB RAM memory, a four-core, four-thread, 64-bit 1.4 GHz ARM-based main processor, four USB 2.0 connectors, a Gigabit Ethernet port, stereo audio jack, Wi-Fi AC and low-energy Bluetooth. The GPU is Broadcom Videocore IV, which is integrated on the main processor die, it has 12 cores, all of them are clocked at 400 MHz, each core is able to run independent instructions, supports a 16-element SIMD vector width, and allows direct memory access DMA. Its theoretical maximum performance is about 24 GFLOPs, whereas a regular laptop computer, featuring an Intel i7 7700HQ yields 63 GFLOPs. Maximum power consumption is 7.5 watts.

3.2.2. Software, operating system for RPI

Debian Raspbian. It is an ARM 32 bits porting of mainstream Debian operating system. This offers compatibility with software packages and techniques, compilers, end-user applications, and provides a key component for this kind of systems: a full-fledged internet browser, the Chromium". PI uses standards monitors and peripherals, and can run in headless mode, that is, no GUI layer is present thus freeing computational resources. Accepts smart clients built in C, C++, Python, and by means of virtual machines is compatible with Java and .NET applications. Debian was selected.

Windows 10 IoT. This OS was also evaluated, but not selected due to some constraints that prevent to achieve some goals. Win10 IoT is a lightweight version of Windows 10, sharing kernel architecture with desktop versions, but lacking a true desktop GUI. Only one UWP (universal windows platform) application can be executed at a given time. Application is developed in regular computers and transferred to RPI by network using specific tools. UWP advantages are rich graphical interface, fast development, strong support for WYSIWIG design, 3 libraries for multithreading with 12 parallel classes, and full support for asynchronous computing. However, because of its UWP-only paradigm, no HTTP server can be deployed. UWP subsystem has significant overhead, that results in somewhat long starting times. At the time of this writing, no stable Windows 10 IoT was available for RPI 3B+ model. Because of these reasons, Windows 10 IoT was not selected for this work.

3.2.3. Software, development and production platform for RPI

HTTP Server. ASP.NET Core 3.0 for ARM32 was selected. Since the middleware must run on small devices, a compact and high-performance server with fast HTTP stack was desirable. ASP.NET is a framework to build web applications and services. NET. Core is an open source version of regular .NET runtime, with a minimal base class library, which was refactored to avoid dependencies, thus reducing size of the runtime itself. Specific libraries, needed for every kind of project, must be installed in design-time using an official package-delivery facility named NuGet, which also deploys specific runtimes, framework libraries and third-party packages with the web application itself. Since libraries are reduced to a bare minimum, and dependencies are avoided, execution time is significatively improved.

Another improvement adopted in this work is the self-contained, cross-platform, in-process HTTP Server, code-named Kestrel. Is a lightweight process that manages web applications with an optimized and complete HTTP stack. Since web application runs in same process space, IPC communications are optimal.

.NET Core 3.0 preview 4 was used in this work. At the time of this writing, version 3.0 is in preview stage, but is stable enough to serve as middleware for this proposal. To run .NET Core apps on RPI, a runtime or SDK must be installed in Raspbian. Runtime is a minimum set of files to support execution. SDK adds programming tools to be executed on RPI command line. NET Core 3.0 for Raspbian allows to two kind of applications: console applications (character-based with no GUI) and ASP.NET web applications. So far, other types of applications are not supported. On RPI, applications can be deployed in two modes. On the one hand, runtime-dependent mode, where a

runtime library must be already present in host OS. On the other hand, self-contained mode, where all runtime files are packed with the application itself, thus avoiding any binary dependency and easing system deployment, at the expense of increased size.

Runtime files are enough to run any ASP.NET Core site or console application, but SDK is used to manage certificates for HTTPs secure transport, to support on-board compilation, and to perform maintenance tasks in the PI itself. For this work, only runtime was deployed to RPI board.

3.2.4 Software, relational database management system for RPI.

*PostgreSQL 9.*6 for Raspbian was selected. Alternatives such as SQL Lite and MySQL were also evaluated.

SQL Lite is an embedded database, that is, a serverless design. Layers of RDBMS are integrated as components of main application, running in the same process; it has a small footprint, averaging 600 KiB, and avoids external dependencies. SQL Lite was not selected because it allows one single write procedure to be executed at a given time, which could easily create a bottleneck. Moreover, since is an in-process database, is not suitable enough for multiuser or networked access.

MySQL is a very fast database engine, with abundant documentation, and important installed base. It provides standard security features with acceptable granularity, and easy replication support. MySQL was no selected because of its with a dual licensing schema (open source and commercial) resulting in some plugins to be available only with the paid version.

PostgreSQL offers many key advantages. It has a very sophisticated optimizer engine. It supports a wide selection of datatypes. Image storage is supported by means of two techniques: (a) storage as binary data arrays (`bytea` datatype) in regular tables, and (b) storage in a hidden, BLOB-optimized table, which is linked by GUIDs to regular tables. It has a large community, with abundant documentation and online resources. Is compatible with many programming languages and offers native connections for many programming platforms. It has a strong support for concurrent writes. Regarding licensing, it is completely free and open source. Supports data science scenarios on ARM V8 architectures, such as the Modular Micro server Data Centre project [4]. Constraints are as follows. In every new connection, for an incoming user, allocates up to 10 MiB memory using a slow procedure. Regarding tablespaces, PostgreSQL is more efficient for writings than for readings, which impacts negatively on scenarios with many read-only transactions. Due to positives aspects, PostgreSQL 9.6 for Raspbian Stretch version was selected for this work.

3.2.5. Software, selection of image processing library for RPI.

In this section, we discuss four alternatives to run OpenCV in RPI devices.

OpenCV 4 for Python 3.5.3, running in middleware. Following a strict sequence, Python can be installed with direct access to OpenCV 4.0. OpenCV functions are accessed in Python by importing the well-known **cv2** object, which exposes all members needed to image processing. Several IDEs are shipped with Raspbian, but none of them features a GUI visual designer.

Native OpenCV accessed from managed code, running in middleware. Accessing compiled OpenCV from other platforms, e.g., a web application, requires an API that exposes OpenCV to each platform. Because there is not an OpenCV porting for .NET Core ARM32, an extensive refactoring of the solution would be needed in order to access Linux objects. As the time of this writing, a Kestrel-specific Linux transport is under development for .NET Core and Redhat Linux [5], it is a library for .NET Core that uses Linux APIs, available in x64, arm64 and arm32 platforms. This would allow managed code to effectively access native functions running Kestrel, the .NET Core HTTP compact server. However, since Raspbian is a Debian Linux and that library was designed for Redhat, extensive testing must be done for Raspberry's ARM32 platform, in order to discard every mismatch that could arise in runtime.

OpenCV compiled for .NET Core on ARM32 Linux, running in middleware. OpenCVSharp is an open-source, cross-platform OpenCV wrapper for .NET framework. It comprises 2 libraries: OpenCV main library, and platform-specific support library. Latter one provides access to underlying OS, and is available for Windows, 64-bit Ubuntu Linux systems, and 64-bit CentOS Linux systems. In our trials, application crashed in RPI when trying to access OpenCV because of lacking an ARM32 porting.

OpenCv.js in HTML5 browser using asm.js or web assembly. With the introduction of HTML5, mainstream vendors quickly updated their browsers in order to support as many features as possible. Nowadays, modern browsers can render online video with few HTML5 tags and can capture webcam streams with WebRTC API [6] to provide real time communications among browsers. To process current image and video data in browser spaces, script execution engine must provide near-native speed. **OpenCv.js** is an extensive Javascript library generated from C/C++ sources, which can be compiled using certain browser built-in facilities. Since Javascript is lingua franca for all known browsers, with this approach, OpenCv.js can be executed almost everywhere. OpenCv.js is a 9MiB file, in minified state. A library with such size, with complex functions and matrix algebra, could not provide proper speed in a browser if a traditional Javascript engine is used. To solve this, OpenCv.js is available in two highly optimized execution models that can be run on browsers so equipped: **asm.js** [7] and **Wasm** (web assembly [8]).

OpenCv.js is obtained as follows. C source code is compiled to an intermediate code, as specified by LLVM (low-level virtual machine) project [9]. LLVM goal is to provide a program representation with platform and language independency. Once in LLVM, a trans compiler called Emscripten [10], takes LLVM bitcode and emits a Javascript *asm.js*, a low-level subset of JavaScript, designed to enhance processing speeds in browsers, by excluding features that cannot be optimized by means of JIT compiler techniques, such as ahead of time compilation to native binary code. Resulting *asm.js* can be compiled with BinaryEn to obtain a web assembly (*wasm*). Wasm is designed to provide Javascript execution engine with native code speed. Is faster than asm.js by generating a shorter, not human-readable, precompiled binary code. Wasm executes in the browser, in a specialized stack machine. It was even reported that, in certain scenarios, Wasm outperforms pure binary code on the same machine [11].

For this work, OpenCv.js was downloaded from [12] and corresponding utilities from [13]. File [12] contains a web assembly version of OpenCV. Both official sites are automatically managed to offer latest stable versions (nightly builds).

3.3 System architecture.

The system has two main blocks, the middleware and the back-end RDBMS. Middleware is built on ASP.NET Core 3.0 preview 4 for Raspbian Linux, provides a set of web pages and a HTML5 REST API. The RDBMS is an instance of PostgreSQL v.9.6. that runs in its own process. **Fig.1** shows components for both blocks. Kestrel server runs in application process and routes every request by resource type. If client requests a webpage, is routed to matching Razor page, if client request a Web API resource, is routed to the controller. Pages and API can be accessed by any process at the same RPI, o from a connected device in the same network. RDBMS can be accessed as any regular PostgreSQL instance.

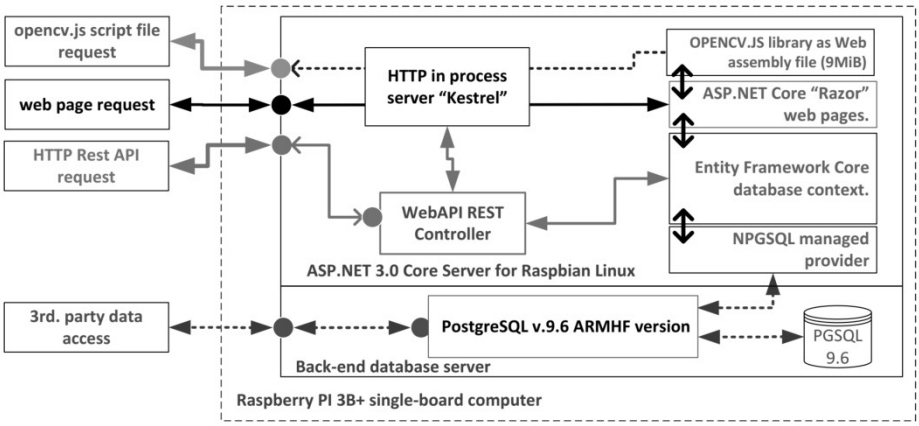

Fig. 1. System components

Since PostgreSQL runs in its own process, external clients can access directly to database. For instance, a PostgreSQL master node can access this way to coordinate replication tasks. Middleware REST API supports CRUD database operations by means of HTTP verbs. For instance, a POST request inserts the object that is serialized in its message body. The following table was created for image storage:

```
Image (id integer, nombre varchar, raster bytea)
```

Image is stored as *byte array* in column "raster". ID is automatically generated, and the name is defined at front-end image acquisition. Web pages can be accessed using any HTML5-compliant browser. System was tested in a regular PC, an RPI board, and an Android 9 smartphone with built-in camera. For industrial applications, a Raspberry server can be deployed, and every device with mainstream browsers can access and use library functions just by accessing a web page. No compilation or installation

procedures are needed at client side, because web pages encapsulate and hide OpenCV complexity using web assemblies. In this way, portability is fully guaranteed.

Fig. 2 shows an example of deployment. The server-side is labeled as *in-house system*, where two RPI boards run middleware and RDBMS respectively, and a third RPI board can run a browser client with an optional webcam. Middleware and RDBMS can run in *headless* mode, with no connected display, keyboard or mouse. In this mode, window manager is not loaded, thus releasing GUI resources to middleware layer. With headless systems, management must be done from a remote computer via a secure terminal, such as PuTTY with SSH support. Remote clients are grouped in *remote access systems* pane, where third-party systems, such desktop applications and server middleware, can access via internet. End user types are the same than in server side. Every end user device must be equipped with an USB or built-in camera, and must be compatible with a HTML5-compliant, Wasm-enabled browser. At the time of this writing, compatible browsers are Mozilla Foundation Firefox, Apple Safari, Google Chrome, Microsoft Edge and Google Chromium (open source).

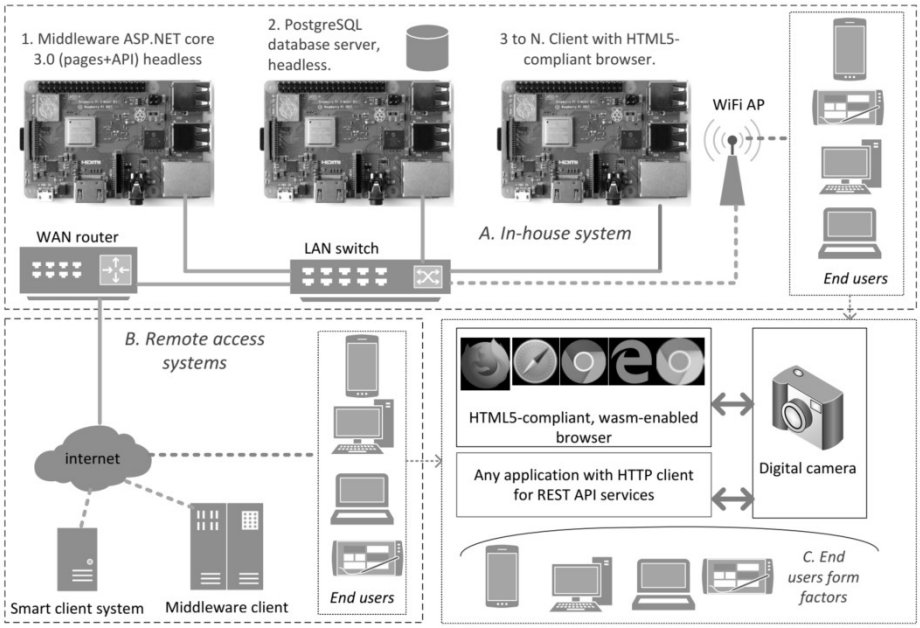

Fig. 2. System deployment example.

4 System benchmarks and results discussion

Two metering techniques were used in order to test the model performance. On the one hand, Javascript Performance API was used to time script events, which provides a resolution of 5ms. On the other hand, Performance Event Recorder (PER) was used in pre-render stages, when scripts are still being loaded. PER is a browser built-in facility, available in all Chromium-based browsers. In the following charts,

measurements were timed in four combinations: single RPI running client + server, Android smartphone client with RPI server, PC running client + server, and a PC client with RPI server. Webcam was a Microsoft® HD-5000® USB camera. PC was a laptop computer with an Intel® i7 7700HQ processor, 32GiB RAM, and 512 GB NVME SSD disk. Network was a standard WLAN b-g-n.

Fig. 3 shows opencv.js script loading and evaluation times, using Performance Event Recorder. In loading times, RPI with client and server was faster than PC, because RPI had to transfer the file locally, while PC must receive the same script file via WLAN. Android figures are not depicted, due to lack of Performance Event Recorder in that platform. After loading, script must be evaluated, and compiled if needed. Script evaluation is performed by the browser, and is a very time-consuming operation, quite noticeably in slow browsers. For instance, Chromium on RPI consumed more than 5 seconds to evaluate opencv.js, whereas Chrome on laptop consumed about 300 ms. Regarding compiling, since opencv.js is delivered in Wasm, it arrives already compiled, so compilation time is negligible, even in RPI (140 ms), and practically unnoticeable in a laptop computer (20 ms).

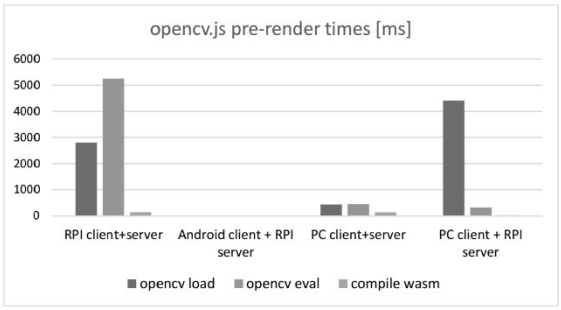

Fig. 4. Opencv.js script pre-render operations.

Fig. 4 shows elapsed times for first and second insertion in databases. Images are always inserted using a POST against Web API, the image is stored in HTTP message body. First insertion implies creation of some objects in memory, which results in a time-consuming operation. However, since our middleware design uses a singleton pattern, data objects creation, such as OR mappers, entities and database connections, occur only once at the server, which gives, as a result, that the same subsequent operations are performed noticeably faster. This was consistent in all client/server combinations, a second operation executed 3 or 4 times faster than first one. From normal operations (that is, after the first one), database insertion is about 800 ms in RPI (client and server), and about 200 ms in PC (client and server). If client connects to server using WLAN, results may vary due to network usage at that time. For instance, in trials, network issues impacted severely in PC times, resulting in figures that are eight times slower than a smartphone client.

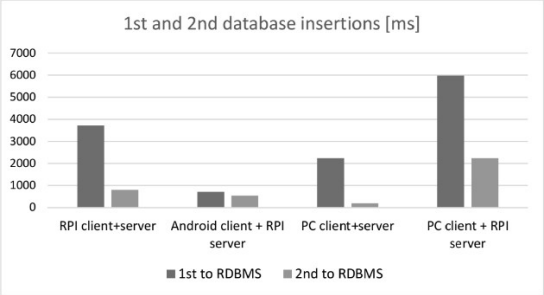

Fig. 4. First and second images insertion in database

Fig. 5 shows elapsed times for two common algorithms, color to grayscale using conversion functions, and edge detection using the well-known Canny algorithm. Since Wasm execution engine lacks automatic garbage collector, unused objects memory must be claimed explicitly in code. However, memory allocation is a very fast operation in OpenCV. In addition, Wasm structure significatively aids Javascript engine optimizer. This results in quite acceptable figures for all platforms. Canny edge detector consumed less than 160 ms in RPI (client and browser), less than 30 ms in smartphone (client), and about 16 ms in a laptop PC, as client, with RPI as server. This turn modern smartphones in very performant devices for custom image processing.

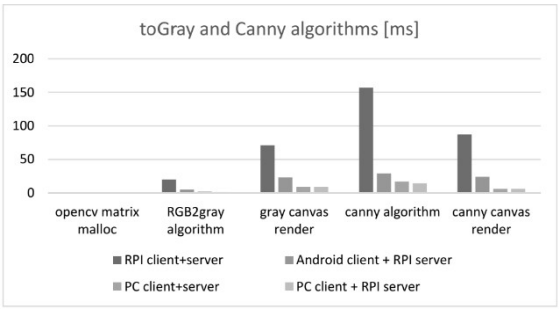

Fig. 5. Common algorithms performance

Fig.5 also show the highly optimized mechanism for memory allocation. Despite lacking a garbage collector, Opencv.js performed consistently, consuming sub-millisecond times when creating matrices, even in RPI boards.

5 Conclusions

In this paper, a distributed system architecture to image processing using Raspberry PI as computing device was presented. The low-cost, low-energy, open source hardware and software, single computer board, running a Raspbian (Linux) operating system are main characteristics of the proposed model.

The application is web-based and uses open internet standards. The middleware is a streamlined open source ASP.NET Core 3.0 web application, with web pages and web API, using an open source PostgreSQL RDBMS for back-end. This layered architecture

provides flexibility to combine client and server components, and since it uses open standards and cross-platform software, it can be deployed on almost every form-factor device, including mobile ones. In addition, it can be combined with any mainstream operating system. Wasm model was designed to be a disruptive change in web applications, and obtained figures support that assertion. Measurements show a very acceptable model performance in resource-constrained devices, such as the Raspberry PI, making this device as an optimal foundation for industrial IoT systems that involve image acquisition and processing.

In consequence, further research must be done to optimize even more this model, perhaps adding a server-side component that complements or aids the client side Wasm script, using a persistent HTTP real time communication facility, such as web-sockets, to obtain a true bi-directional distributed system to image processing, with adaptive load balance.

References

[1] A. J. Trappey, C. V. Trappey, U. H. Govindarajan, A. C. Chuang and J. J. Sun, "A review of essential standards and patent landscapes for the Internet of Things: A key enabler for Industry 4.0," *Advanced Engineering Informatics,* vol. 33, pp. 208-229, 2017.

[2] I. Culjak, D. Abram, T. Pribanic, D. Hrvoje and M. Cifrek, "A brief introduction to OpenCV," in *2012 Proceedings of the 35th International Convention MIPRO*, Opatija, Croatia, 2012.

[3] H. Fassold, "Computer Vision on the GPU – Tools, Algorithms and Frameworks," in *IEEE International Conference on Intelligent Engineering Systems*, Budapest, 2016.

[4] Oleksiak, Ariel; Donoghue, Andrew;, "Modular Microserver Data Centre," M2DC, 30 12 2018. [Online]. Available: https://m2dc.eu/en/get-m2dc-white-paper/. [Accessed 30 04 2019].

[5] T. Deseyn, "GitHub - redhat-developer/kestrel-linux-transport: Linux Transport for Kestrel," Github, [Online]. Available: https://github.com/redhat-developer/kestrel-linux-transport. [Accessed 30 04 2019].

[6] Web Realtime Communications Organization, "Web RTC native APIs," Google, Inc., [Online]. Available: https://webrtc.org/native-code/native-apis/. [Accessed 30 4 2019].

[7] D. Herman, L. Wagner and A. Zakai, "asm.js - an extraordinarily optimizable, low-level subset of JavaScript," Mozilla.org, 18 08 2014. [Online]. Available: http://asmjs.org/. [Accessed 2019 04 30].

[8] A. Haas, A. Rossberg, D. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai and J. F. Bastien, "Bringing the Web up to Speed with WebAssembly," in *ACM SIGPLAN Programming Language Design and Implementation - PLDI 2017*, Barcelona, 2017.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization. CGO 2004.*, San Jose, CA. USA., 2004.

[10] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," in *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*, Portland, OR. USA., 2011.

[11] B. Malle, N. Giuliani, P. Kieseberg and A. Holzinger, "The need for speed of AI applications. Performance comparison of native vs. browser-based algorithm implementations," 02 2018. [Online]. Available: https://arxiv.org/pdf/1802.03707.pdf. [Accessed 30 04 2019].

[12] OpenCV.org, "opencv.js," [Online]. Available: https://docs.opencv.org/master/opencv.js. [Accessed 30 04 2019].

[13] Opencv.org, "utils.js," [Online]. Available: https://docs.opencv.org/master/utils.js. [Accessed 30 04 2019].

[14] R. Lienhart and J. Maydt, "An Extended Set of Haar-like Features for Rapid Object Detection," in *IEEE International Conference on Image Processing 2002*, 2002.

[15] S. Taheri, A. Veindenbaum, A. Nicolau and M. R. Haghighat, "OpenCV.js: Computer Vision Processing for the Web," Center for Embedded and Cyber-Physical Systems. University of California, Irvine., Irvine, 2017.